Weakest Precondition Calculus, Revisited using Why3

Claude Marché, Asma Tafat

Weakest Precondition Calculus, Revisited using Why3

Claude Marché, Asma Tafat

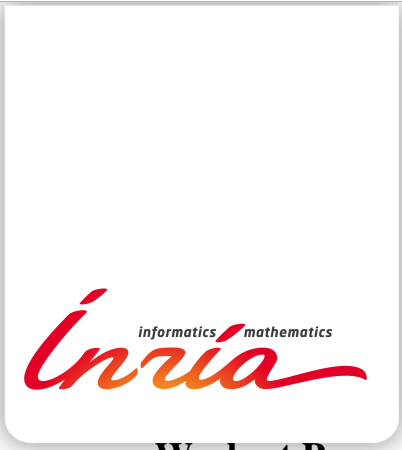
RESEARCH REPORT

N° 8185

December 2012

Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--8185--FR+ENG



Weakest Precondition Calculus, Revisited using Why3

Claude Marché*†, Asma Tafat†*

Project-Team Toccata

Research Report n° 8185 — December 2012 — 32 pages

Abstract: This report has two objectives. First, we present an original method of proof of soundness of a weakest precondition calculus, based on the notion of *blocking semantics*. The method mimics, at the level of logic specifications, the classical proof of *type soundness*. Moreover, the proof is performed formally using the Why3 environment for deductive verification, and we illustrate, along the development of the case study, the advanced features of Why3 we used. The result is a revisited presentation the weakest precondition calculus which is easy to follow, although formally made, thanks in particular to the high degree of proof automation that allows us to focus on the key points.

Key-words: Deductive Verification, Weakest Precondition Calculus, Blocking Semantics, Automated Provers, Coq Proof Assistant

Cette étude a été partiellement financée par le projet U3CAT (ANR-08-SEGI-021, http://frama-c.com/u3cat/) de l'Agence Nationale de la Recherche et le projet Hi-lite (http://www.open-do.org/projects/hi-lite/) du pôle de compétitivité Systematic de la région Île-de-France.

* INRIA Saclay - Île-de-France
† Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Calcul de plus faible précondition, revisité en Why3

Résumé : Ce rapport a deux objectifs. D'une part, nous présentons une méthode originale de preuve de correction d'un calcul de plus faible précondition, fondée sur la notion de *sémantique bloquante*. La méthode imite, au niveau des spécifications logiques, la méthode classique de preuve de *type soundness*. D'autre part, cette preuve est réalisée formellement dans l'environnement de vérification déductive Why3, et nous illustrons, au fur et à mesure du développement de cette étude de cas, les fonctionnalités avancées de Why3 que nous avons utilisées. Le résultat constitue une présentation revisitée du calcul de plus faible précondition, et qui, bien qu'elle soit réalisée formellement, est facile à suivre, grâce en particulier au haut degré d'automatisation des preuves qui permet de se focaliser sur les points clés.

Mots-clés : Vérification déductive, calcul de plus faible pré-condition, sémantique bloquante, prouveurs automatiques, assistant de preuve Coq

Contents

List of Figures

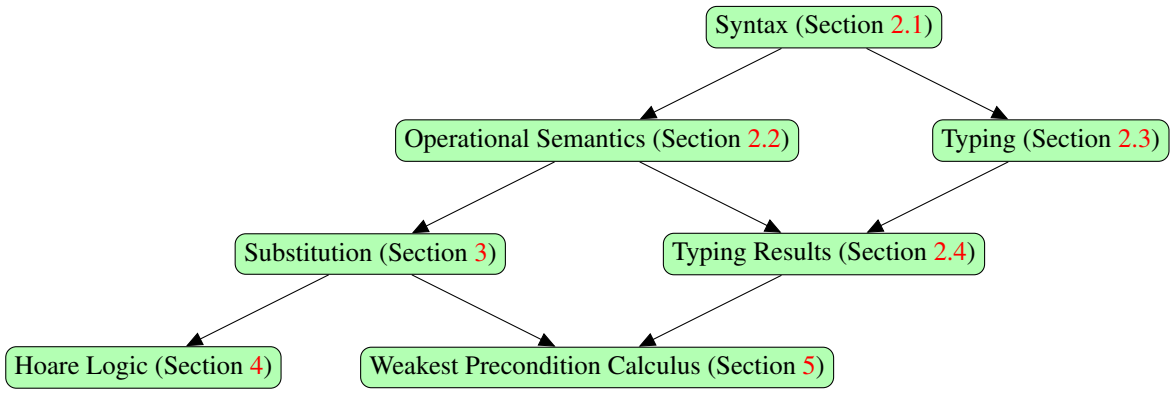


Figure 1: Structure of the development

1 Introduction

The Why3 system [7] is an environment for *deductive program verification*. It allows the user to write programs annotated with logical specifications, and then generate proof obligations that guarantee the functional correctness. Why3 can submit these obligations to a large collection of provers: automatic ones such as *SMT solvers* (Alt-Ergo [5], CVC3 [3], Z3 [9], etc.) or interactive proof assistants like Coq [4] or PVS [11].

In the spectrum of deductive verification tools, Why3 positions itself in the middle of interactive environments, that offer highly expressive languages but a low level of automation (Coq, PVS, Isabelle/HOL, etc.), and more automated systems but equipped with poorer languages (Spec#, VCC, Frama-C, KeY, etc.). A natural objective is to catch the best possible between these extremes: a sufficiently expressive language that still permits significantly automated proofs.

A first objective of this report is to illustrate the capabilities of Why3 on some case study, that involves complex objects like recursive algebraic data types and inductive predicate definitions, so as to illustrate how proofs can be automatized in such a context. A class of examples involving such kind of objects is given by the symbolic computation methods, in particular the tools for analyzing programs: compilers, static analyzers, etc. We chose to formalize a calculus of weakest precondition, partially following a recent approach due to Herms *et al.* [10] for the certification of a verification condition generator, using Coq. This new approach is based on a notion of *big-step blocking semantics*, defined co-inductively. The second objective of this work is to present an original approach of the weakest precondition calculus. The originality resides in the use of a *small-steps* blocking semantics, which is moreover formalized using the Why3 language, significantly restricted with respect to those of Coq.

Apart from the algebraic data types and the inductive predicates, the advances features of Why3 that we are going to use concern the tools available to build proofs. First, we use a *transformation* of Why3 to make structural inductions. Second, when the goals to prove get too complex to be proved fully automatically, we call the Coq back-end, and within Coq we use the « `why3` » *tactic* that can finish the proof of a sub-goal by calling automated provers through Why3. These features allow us to make in Coq only the subtle parts of the proofs.

Our case study is structured into components, named *theories* in Why3, and this report follows the same structure, represented on Figure 1. In Section 2 we define the language under study: its syntax (Section 2.1), its operational semantics (Section 2.2) and its typing rules (Section 2.3). In Section 2.4 we formulate and prove classical results of preservation of typing by reduction. Section 3 is devoted to the notion of substitution and to the important problems related to fresh variables. Although not needed for the remaining, Section 4 presents the classical Hoare logic rules and prove their soundness. The main results are then presented in Section 5: definition of the weakest precondition calculus, and soundness results based on the small-steps blocking semantics.

2 Formalization of Toy Imperative Language

This section formalize our language which contains assignment, sequence, a conditional statement and a while loop.

For each of the theories we will present, we are going to tell how the proof of the lemmas are obtained, by

giving the table that is automatically generated by Why3 from the proof sessions [6].

In most cases, provers are launched with a time limit of 5 seconds and a memory limit of 1 gigabyte. The cases where either one of these limits is reach are indicated in the tables below by results between parentheses.

2.1 Syntax

The first theory defines our language syntax. Traditionally, one defines language syntax with grammars. Here, we directly give Why3 definitions under the form of algebraic data type declarations. We can not present the Why3 syntax in detail in this report, you may refer to the manual [6] or J.-C.Filliâtre lecture given at JFLA 2012[1]. The syntax is similar to the OCaml one, we hope it is intuitive enough for the reader.

Our language has three base types which correspond to three sets of values.

```
type datatype = TYunit | TYint | TYbool      (** basic types unit, int and bool *)
type value = Vvoid | Vint int | Vbool bool   (** corresponding sets of values *)
```

We introduce *abstract* types for identifiers, by making a distinction between identifiers for mutable variables, and identifiers for logic variables.

```
type mident  (** identifiers for mutable variables *)
type ident   (** identifiers for logic variables *)
```

The terms of the logic language contain values, logic variables, dereferencing of mutable variables, and binary operations $+$, $-$, $\times$ and $\leq$ (other operators could be added without additional difficulties).

```
type operator = Oplus | Ominus | Omult | Ole  (** operators +, -, *, ≤ *)
type term =                                   (** Terms *)
  | Tvalue value                              (** values *)
  | Tvar ident                                (** logic variables *)
  | Tderef mident                             (** access to mutable variables *)
  | Tbin term operator term                   (** binary operations *)
```

Formulas of the logic language include terms (if Boolean), classical connectors for conjunction, negation and implication, and universal quantification. These are the only needed connectors, but others could be added. We also add local binding with `let` that allows us to write more elegant rules in the weakest precondition calculus. Note that we do not use any advanced technique such as De Bruijn indexes for local bindings. Difficulties related to variable naming will be handled in Section 3.

```
type fmla =                       (** Formulas *)
  | Fterm term                    (** terms: atomic formulas *)
  | Fand fmla fmla                (** conjunction *)
  | Fnot fmla                     (** negation *)
  | Fimplies fmla fmla            (** implication *)
  | Flet ident term fmla          (** local binding: let id = term in fmla *)
  | Fforall ident datatype fmla   (** universal quantification: forall id : ty, fmla *)
```

Finally, the syntax of statements of our imperative language follows.

```
type stmt =                 (** Statements *)
  | Sskip                   (** no-op statement *)
  | Sassign mident term     (** assignment id := term *)
  | Sseq stmt stmt          (** sequence *)
  | Sif term stmt stmt      (** conditional statement *)
  | Sassert fmla            (** assertion statement *)
  | Swhile term fmla stmt   (** while loop with condition, invariant and body *)
```

[1] http://why3.lri.fr/jfla-2012/

We consider assignment of a mutable variable by a term, sequences (`skip` constant representing an empty sequence), standard conditional statement and a *while* loop.

Unconventionally, we add logic annotations directly in the syntax: explicit invariants for loops, as well as assert statements.

This first theory contains only definitions and no lemmas. Therefore, there is nothing to prove.

2.2 Operational semantics

The aim of the second theory is to define the operational semantics of our language. We choose a small step version.

In a first step, we define evaluation environments: for mutable variable, on one hand, and for logic variables on the other.

Type `env` denotes environments for mutable variables. It is a map that associates a value to any `mident`. For this we use the Why3 standard library generic type `map`.

```
use export Syntax
use map.Map as IdMap
type env = IdMap.map mident value     (** map global mutable variables to their value *)
function get_env (i:mident) (sigma:env) : value = IdMap.get sigma i
```

`get_env` function provides a shortcut for accessing to the elements of an environment.

Type `stack` denotes environments for logic variables: a stack of pairs (`ident`, `value`). We use the generic type `list` of Why3 standard library.

```
use export list.List
type stack = list (ident, value)     (** map local immutable variables to their value *)
function get_stack (i:ident) (pi:stack) : value =
  match pi with
  | Nil → Vvoid
  | Cons (x,v) r → if x=i then v else get_stack i r
  end
```

`get_stack` function provides a shortcut to access to an element; it is defined by induction on the list. Note that this function returns `void` if we try to access to a variable absent from the list. This case never happen on well-typed programs as defined in the next section. Finally, we pose lemmas showing a similar behavior to maps, that are automatically proved.

```
lemma get_stack_eq: forall x:ident, v:value, pi:stack.
  get_stack x (Cons (x,v) pi) = v
lemma get_stack_neq: forall x i:ident, v:value, pi:stack.
  x ≠ i → get_stack i (Cons (x,v) pi) = get_stack i pi
```

Next, we define an auxiliary function for evaluating binary operations. This definition is made by pattern-matching on the values and the operator. Since in Why3 functions must be total, we complete our definition by returning `void` in ill-typed cases. [2].

```
function eval_bin (x:value) (op:operator) (y:value) : value =
  match x,y with
  | Vint x,Vint y →
     match op with
     | Oplus  → Vint (x+y)
     | Ominus → Vint (x-y)
     | Omult  → Vint (x*y)
     | Ole    → Vbool (if x ≤ y then True else False)
     end
  | _,_ → Vvoid
  end
```

[2]Note that `True` and `False` are the constructors of Boolean type of Why3.

We can now define evaluation of terms as a total recursive function that returns a `value`, and evaluation of formulas by a total recursive predicate. These two functions take a *current state* of a program, given by two environments Σ and Π, as arguments.

```
function eval_term (sigma:env) (pi:stack) (t:term) : value =
  match t with
  | Tvalue v       → v
  | Tvar id        → get_stack id pi
  | Tderef id      → get_env id sigma
  | Tbin t1 op t2  → eval_bin (eval_term sigma pi t1) op (eval_term sigma pi t2)
  end

predicate eval_fmla (sigma:env) (pi:stack) (f:fmla) =
  match f with
  | Fterm t            → eval_term sigma pi t = Vbool True
  | Fand f1 f2         → eval_fmla sigma pi f1 ∧ eval_fmla sigma pi f2
  | Fnot f             → not (eval_fmla sigma pi f)
  | Fimplies f1 f2     → eval_fmla sigma pi f1 → eval_fmla sigma pi f2
  | Flet x t f         → eval_fmla sigma (Cons (x,eval_term sigma pi t) pi) f
  | Fforall x TYint f  → forall n:int. eval_fmla sigma (Cons (x,Vint n) pi) f
  | Fforall x TYbool f → forall b:bool. eval_fmla sigma (Cons (x,Vbool b) pi) f
  | Fforall x TYunit f → eval_fmla sigma (Cons (x,Vvoid) pi) f
  end
```

Note that in the case of binders `Flet` and `Fforall`, the linked variable `x` is added to the stack. In the following, we will note these functions in the form $[\![t]\!]_{\Sigma,\Pi}$ and $[\![f]\!]_{\Sigma,\Pi}$.

Finally, we define the notion of *valid* formula, in the sense that it is true in any state.

```
predicate valid_fmla (p:fmla) = forall sigma:env, pi:stack. eval_fmla sigma pi p
```

Execution of a statement in a given state is classically defined by a one step reduction relation noted

$$\Sigma, \Pi, s \rightsquigarrow \Sigma', \Pi', s'$$

Since there are statements that do not reduce, we cannot formalize this relation by a Why3 function[3]. Instead, we use an *inductive predicate* of Why3[4]. Our definition is a formalization close to the usual representation using inference rules. It is shown on Figure 2. Thus, the `one_step_while_true` case correspond to the rule

$$\frac{[\![inv]\!]_{\Sigma,\Pi} \qquad [\![cond]\!]_{\Sigma,\Pi} = True}{\Sigma, \Pi, \texttt{while}\ cond\ \texttt{invariant}\ inv\ \texttt{do}\ body \quad \rightsquigarrow \quad \Sigma, \Pi, body; \texttt{while}\ cond\ \texttt{invariant}\ inv\ \texttt{do}\ body}$$

We use a *blocking* semantic in the way proposed by Herms *et al.* [10]: The execution blocks whenever an invalid assertion is met or if a loop invariant is broken when the loop condition is checked (as shown in the rule above). Conversely, assuming that the program is well-typed, the only reason why a statement would not execute is that one of its annotations is not respected.

The advantage of such a definition is that we define the fact that *a program respect its specifications* by the fact that *it executes without blocking* [10], and this applies in particular for a program does not terminate.

Another inductive predicate defines the execution in n steps of a program, by a reflexive-transitive closure, shown on Figure 3. Remark that we make explicit the number n of steps of execution in this definition, to do proofs by induction on n. Notice that the built-in `int` type in Why3 is the type of integers (signed), there is no built-in type for natural numbers only. Thus, to make well-founded inductions, we pose the following elementary, but essential, lemma.

```
lemma steps_non_neg: forall sigma1 sigma2:env, pi1 pi2:stack, s1 s2:stmt, n:int.
  many_steps sigma1 pi1 s1 sigma2 pi2 s2 n → n ≥ 0
```

[3] We remind that Why3 functions are always total.

[4] The notion of inductive predicate in Why3 is similar to Coq or PVS ones: It is the smallest relation verifying the given clauses. Why3 verifies the usual monotonicity conditions for such definitions.

```
inductive one_step env stack stmt env stack stmt =

  | one_step_assign : forall sigma sigma':env, pi:stack, x:mident, t:term.
      sigma' = IdMap.set sigma x (eval_term sigma pi t) →
      one_step sigma pi (Sassign x t) sigma' pi Sskip

  | one_step_seq_noskip: forall sigma sigma':env, pi pi':stack, s1 s1' s2:stmt.
      one_step sigma pi s1 sigma' pi' s1'              →
      one_step sigma pi (Sseq s1 s2) sigma' pi' (Sseq s1' s2)

  | one_step_seq_skip: forall sigma:env, pi:stack, s:stmt.
      one_step sigma pi (Sseq Sskip s) sigma pi s

  | one_step_if_true: forall sigma:env, pi:stack, t:term, s1 s2:stmt.
      eval_term sigma pi t = Vbool True                →
      one_step sigma pi (Sif t s1 s2) sigma pi s1

  | one_step_if_false: forall sigma:env, pi:stack, t:term, s1 s2:stmt.
      eval_term sigma pi t = Vbool False               →
      one_step sigma pi (Sif t s1 s2) sigma pi s2

  | one_step_assert: forall sigma:env, pi:stack, f:fmla.
      eval_fmla sigma pi f                             →       (** blocking semantics *)
      one_step sigma pi (Sassert f) sigma pi Sskip

  | one_step_while_true: forall sigma:env, pi:stack, cond:term, inv:fmla, body:stmt.
      eval_fmla sigma pi inv                           →       (** blocking semantics *)
      eval_term sigma pi cond = Vbool True             →
      one_step sigma pi (Swhile cond inv body) sigma pi (Sseq body (Swhile cond inv body))

  | one_step_while_false: forall sigma:env, pi:stack, cond:term, inv:fmla, body:stmt.
      eval_fmla sigma pi inv                           →       (** blocking semantics *)
      eval_term sigma pi cond = Vbool False            →
      one_step sigma pi (Swhile cond inv body) sigma pi Sskip
```

Figure 2: One-step reduction

```
inductive many_steps env stack stmt env stack stmt int =

  | many_steps_refl: forall sigma:env, pi:stack, s:stmt.
      many_steps sigma pi s sigma pi s 0

  | many_steps_trans: forall sigma1 sigma2 sigma3:env, pi1 pi2 pi3:stack,
                             s1 s2 s3:stmt, n:int.
      one_step sigma1 pi1 s1 sigma2 pi2 s2 → many_steps sigma2 pi2 s2 sigma3 pi3 s3 n →
      many_steps sigma1 pi1 s1 sigma3 pi3 s3 (n+1)
```

Figure 3: Many steps of reduction

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `get_stack_eq`	0.01	0.01	0.03	0.02		(5s)	(5s)
lemma `get_stack_neq`	0.02	0.01	0.03	0.02		(5s)	(5s)
lemma `steps_non_neg`					0.53		

Figure 4: Proof results for theory `Operational Semantics`

To end this theory, we define the notion of reducibility of a program in a given state.

```
predicate reducible (sigma:env) (pi:stack) (s:stmt) =
  exists sigma':env, pi':stack, s':stmt. one_step sigma pi s sigma' pi' s'
```

So, for the case of our theory modeling the operational semantics, there are 3 lemmas to prove. Results are shown on Figure 4. As expected, the lemma `steps_non_neg` cannot be proved by automated provers, as it requires an induction on the `many_steps` predicate. We do the proof in Coq, and this proof is easy (one line):

```
induction 1; auto with zarith.
```

2.3 Typing

This theory introduces typing of programs. At first, we define a total function that returns a type of some value.

```
function type_value (v:value) : datatype =
  match v with
    | Vvoid   → TYunit
    | Vint _  →  TYint
    | Vbool _ → TYbool
  end
```

Type of the binary operators of our language is given by a simple inductive predicate.

```
inductive type_operator (op:operator) (ty1 ty2 ty: datatype) =
  | Type_plus  : type_operator Oplus TYint TYint TYint
  | Type_minus : type_operator Ominus TYint TYint TYint
  | Type_mult  : type_operator Omult TYint TYint TYint
  | Type_le    : type_operator Ole TYint TYint TYbool
```

We define typing environments similarly to evaluation ones.

```
type type_stack = list (ident, datatype)
    (** map local immutable variables to their type *)
function get_vartype (i:ident) (pi:type_stack) : datatype =
  match pi with
  | Nil → TYunit
  | Cons (x,ty) r → if x=i then ty else get_vartype i r
  end
type type_env = IdMap.map mident datatype
    (** map global mutable variables to their type *)
function get_reftype (i:mident) (e:type_env) : datatype = IdMap.get e i
```

Typing judgments for terms, formulas and statements are, then, naturally defined by new inductive predicates, given on Figures 5, 6 and 7

```
inductive type_term type_env type_stack term datatype =

  | Type_value : forall sigma: type_env, pi:type_stack, v:value.
      type_term sigma pi  (Tvalue v) (type_value v)

  | Type_var : forall sigma: type_env, pi:type_stack, v: ident, ty:datatype.
      (get_vartype v pi = ty) →  type_term sigma pi (Tvar v) ty

  | Type_deref : forall sigma: type_env, pi:type_stack, v: mident, ty:datatype.
      (get_reftype v sigma = ty) → type_term sigma pi (Tderef v) ty

  | Type_bin : forall sigma: type_env, pi:type_stack, t1 t2 : term, op:operator,
      ty1 ty2 ty:datatype.
      type_term sigma pi t1 ty1 → type_term sigma pi t2 ty2 →
      type_operator op ty1 ty2 ty → type_term sigma pi (Tbin t1 op t2) ty
```

Figure 5: Typing rules for terms

```
inductive type_fmla type_env type_stack fmla =

  | Type_term : forall sigma: type_env, pi:type_stack, t:term.
      type_term sigma pi t TYbool → type_fmla sigma pi (Fterm t)

  | Type_conj : forall sigma: type_env, pi:type_stack, f1 f2:fmla.
      type_fmla sigma pi f1 → type_fmla sigma pi f2 → type_fmla sigma pi (Fand f1 f2)

  | Type_neg : forall sigma: type_env, pi:type_stack, f:fmla.
      type_fmla sigma pi f → type_fmla sigma pi (Fnot f)

  | Type_implies : forall sigma: type_env, pi:type_stack, f1 f2:fmla.
      type_fmla sigma pi f1 → type_fmla sigma pi f2 →
      type_fmla sigma pi (Fimplies f1 f2)

  | Type_let : forall sigma: type_env, pi:type_stack, x:ident, t:term, f:fmla, ty:datatype.
      type_term sigma pi t ty → type_fmla sigma (Cons (x,ty) pi) f →
      type_fmla sigma pi (Flet x t f)

  | Type_forall : forall sigma: type_env, pi:type_stack, x:ident, f:fmla, ty:datatype.
      type_fmla sigma (Cons (x,ty) pi) f → type_fmla sigma pi (Fforall x ty f)
```

Figure 6: Typing rules for formulas

```
inductive type_stmt type_env type_stack stmt =

  | Type_skip : forall sigma: type_env, pi:type_stack. type_stmt sigma pi Sskip

  | Type_seq : forall sigma: type_env, pi:type_stack, s1 s2:stmt.
      type_stmt sigma pi s1 → type_stmt sigma pi s2 →
      type_stmt sigma pi (Sseq s1 s2)

  | Type_assigns : forall sigma: type_env, pi:type_stack, x:mident, t:term, ty:datatype.
      (get_reftype x sigma = ty) → type_term sigma pi t ty →
      type_stmt sigma pi (Sassign x t)

  | Type_if : forall sigma: type_env, pi:type_stack, t:term, s1 s2:stmt.
      type_term sigma pi t TYbool → type_stmt sigma pi s1 → type_stmt sigma pi s2 →
      type_stmt sigma pi (Sif t s1 s2)

  | Type_assert : forall sigma: type_env, pi:type_stack, p:fmla.
      type_fmla sigma pi p → type_stmt sigma pi (Sassert p)

  | Type_while : forall sigma: type_env, pi:type_stack, cond:term, body:stmt, inv:fmla.
      type_fmla sigma pi inv → type_term sigma pi cond TYbool →
      type_stmt sigma pi body → type_stmt sigma pi (Swhile cond inv body)
```

Figure 7: Typing rules for statements

2.4 Relations between typing and operational semantic

This theory contains our first results linking typing and execution. At first, it defines a predicate for compatibility between typing and evaluation environments, that is, to each identifier of type t is associated a value of type t.

```
predicate compatible_env (sigma:env) (sigmat:type_env) (pi:stack) (pit: type_stack) =
  (forall id: mident. type_value (get_env id sigma) = get_reftype id sigmat) ∧
  (forall id: ident. type_value (get_stack id pi) = get_vartype id pit)
```

Then we pose an inversion lemma, that allows us to know which is the constructor of a value, from its type.

```
lemma type_inversion : forall v:value.
   match (type_value v) with
    | TYbool → exists b: bool. v = Vbool b
    | TYint  → exists n: int. v = Vint n
    | TYunit → v = Vvoid
  end
```

The results of proofs of all the lemmas in this theory are given on Figure 8. For this particular lemma, an obvious case-based reasoning allows to prove such lemma. Nevertheless, theorem provers are not able to prove this goal[5]. We could do this proof with Coq, but we can also use Why3 features: goals *transformations*. In Why, there exists such a transformation to make structural induction. Here, the type `value` is not recursive, so it amounts to a case-based reasoning. The three generated sub-goals are automatically discharged.

We also made a Coq proof of `type_inversion` lemma. It can be done easily thanks to another feature of Why3: the `why3` *tactic* of Coq. It is a tactic loaded as Coq *plug-in* by

```
Require Import Why3.
Ltac ae := why3 "alt-ergo" timelimit 5
```

[5]Indeed, we realized later in the development that CVC3 2.2 proves that lemma

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `type_inversion`	0.03	0.02	0.04	(5s)	1.02	(5s)	(5s)
transformation `induction_ty_lex`							
subgoal 1	0.08	0.05	0.04	(5s)		0.07	0.07
lemma `eval_type_term`							
transformation `induction_ty_lex`							
transformation `split_goal_wp`							
subgoal 1	0.50	0.13	0.18	0.23		(5s)	(5s)
subgoal 2	0.62	0.14	0.27	0.31		(5s)	(5s)
subgoal 3	0.66	0.22	0.06	0.07		(5s)	(5s)
subgoal 4	(5s)	(5s)	(5s)	(5s)	4.14	(5s)	(5s)
lemma `type_preservation`					7.04		

Figure 8: Proof Results for theory `TypingAndSemantics`

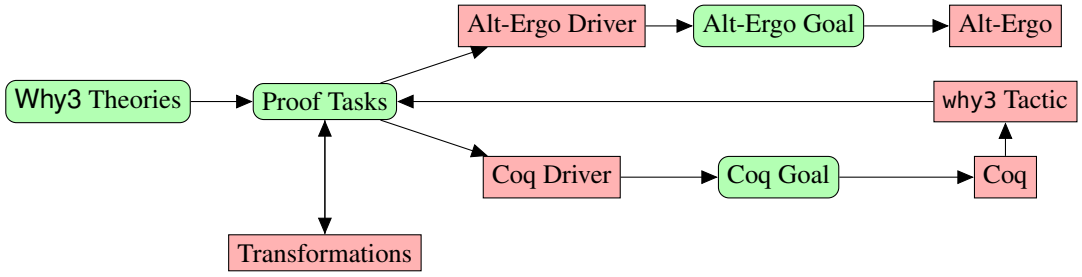


Figure 9: Architecture of Why3 and `why3` tactic of Coq.

The tactic `ae` is therefore a shortcut to call Alt-Ergo from Coq, with a timeout fixed to 5 seconds. The tactic allows, then, to call (through Why3) theorem provers, according to the architecture of Figure 9.

The proof of `type_inversion` lemma is then

```
destruct v; ae.
```

It should be noted that this tactic does not produce any proof term, so we must trust both the way it reconstructs a Why3 task from the current Coq goal, and theorem provers called after, the same way we trust these theorem provers when called directly from Why3.

The first significant result states that a well-typed term of type t, is evaluated to a value of type t, too.

```
lemma eval_type_term:
  forall t:term, sigma:env, pi:stack, sigmat:type_env, pit:type_stack, ty:datatype.
    compatible_env sigma sigmat pi pit →
    type_term sigmat pit t ty          → type_value (eval_term sigma pi t) = ty
```

This lemma has to be proved by structural induction on t. Once again, instead of making proof entirely in Coq, we first apply Why3 transformation for structural induction, which produces 4 new sub-goals, corresponding to a standard induction schema. Among these sub-goals, 3 are automatically proved (See Figure 8), the only one that we have to prove with Coq concerns binary operations which is done in 7 lines:

```
destruct t; auto.
simpl; intros.
inversion H2; subst; clear H2.
destruct H9 as (h1 & h2 & h3).
generalize (type_inversion (eval_term sigma pi t1)).
generalize (type_inversion (eval_term sigma pi t2)).
destruct h3; ae.
```

The second result is preservation of well-typing by reduction.

```
lemma type_preservation : forall s1 s2:stmt, sigma1 sigma2:env, pi1 pi2:stack,
                                 sigmat:type_env, pit:type_stack.
   type_stmt sigmat pit s1 ∧ compatible_env sigma1 sigmat pi1 pit ∧
   one_step sigma1 pi1 s1 sigma2 pi2 s2 →
   type_stmt sigmat pit s2 ∧ compatible_env sigma2 sigmat pi2 pit
```

This lemma is proved in Coq, by induction on `one_step` hypothesis. We obtain 8 sub-cases, 7 of them are automatically discharged by `why3` tactic. The last sub-goal, the sequence, requires the hypothesis of well-typedness of the sequence before calling `why3` tactic.

```
intros s1 s2 sigma1 sigma2 pi1 pi2 sigmat pit (h1,(h2,h3)).
induction h3; try ae.
inversion h1; subst; clear h1; ae.
```

In total, this proof of preservation of typing by reduction is only three lines long in Coq!
Notice that 5 of the 7 seconds (see Figure 8) are lost when waiting for Alt-Ergo failing on the 8th sub-goal.

3 Substitutions, fresh variables

For defining our weakest precondition calculus, we will need to substitute variables. We dedicate a specific theory to the operation of substitution and its properties, that are particularly expressed under hypothesis of freshness of variables. Naming and freshness problems are classically difficult points to handle when we formalize languages with binders, as shown by the famous *POPLmark* challenge [1]. We propose here a basic approach, that we believe is easy to deal with, and sufficient for our study.

First, we define the substitution operation, where we do not care about the problem of variable capture. The only one operation that we are interested in is to substitute a mutable program variable by a logical variable.

```
(** substitution of a mutable variable [x] by a logic variable [v]
    warning: proper behavior only guaranteed if [v] is fresh *)
function msubst_term (t:term) (x:mident) (v:ident) : term =
  match t with
  | Tvalue _ | Tvar _ → t
  | Tderef y          → if x = y then Tvar v else t
  | Tbin t1 op t2     → Tbin (msubst_term t1 x v) op (msubst_term t2 x v)
  end
```

```
function msubst (f:fmla) (x:mident) (v:ident) : fmla =
  match f with
  | Fterm e        → Fterm (msubst_term e x v)
  | Fand f1 f2     → Fand (msubst f1 x v) (msubst f2 x v)
  | Fnot f         → Fnot (msubst f x v)
  | Fimplies f1 f2 → Fimplies (msubst f1 x v) (msubst f2 x v)
  | Flet y t f     → Flet y (msubst_term t x v) (msubst f x v)
  | Fforall y ty f → Fforall y ty (msubst f x v)
  end
```

Then, we define the notion of *fresh* variable, that is, does not appear in a term or a formula. These are natural recursive definitions.

```
(** [fresh_in_term id t] is true when [id] does not occur in [t] *)
predicate fresh_in_term (id:ident) (t:term) =
  match t with
  | Tvalue _ | Tderef _ → true
  | Tvar i       → id ≠ i
  | Tbin t1 _ t2 → fresh_in_term id t1 ∧ fresh_in_term id t2
  end
```

```
predicate fresh_in_fmla (id:ident) (f:fmla) =
  match f with
  | Fterm e                    → fresh_in_term id e
  | Fand f1 f2 | Fimplies f1 f2 → fresh_in_fmla id f1 ∧ fresh_in_fmla id f2
  | Fnot f                     → fresh_in_fmla id f
  | Flet y t f                 → id ≠ y ∧ fresh_in_term id t ∧ fresh_in_fmla id f
  | Fforall y ty f             → id ≠ y ∧ fresh_in_fmla id f
  end
```

For reasoning about substitutions in the remaining of our formalization, we need some general lemmas we present now. Unsurprisingly, these lemmas have freshness hypothesis on concerned variables.

The two first lemmas relate the evaluation of a term (resp. a formula) on which we apply a substitution, with the evaluation of this term (resp. formula) in a modified state. In other words we have the identity

$$[\![t[x \leftarrow v]]\!]_{\Sigma,\Pi} = [\![t]\!]_{\Sigma[x \leftarrow \Pi(v)],\Pi}$$

```
lemma eval_msubst_term: forall e:term, sigma:env, pi:stack, x:mident, v:ident.
  fresh_in_term v e →
  eval_term sigma pi (msubst_term e x v) =
    eval_term (IdMap.set sigma x (get_stack v pi)) pi e

lemma eval_msubst: forall f:fmla, sigma:env, pi:stack, x:mident, v:ident.
  fresh_in_fmla v f →
  (eval_fmla sigma pi (msubst f x v) ↔
    eval_fmla (IdMap.set sigma x (get_stack v pi)) pi f)
```

These lemmas are proved by structural induction on the term (resp. formula), thanks to the Why3 transformation. All sub-goals are automatically discharged (see Figure 10), except two which must be proved in Coq (case `Flet` and `Fforall`). These proofs are simple:

```
destruct f; auto.
simpl; ae.
```

The next lemmas allow us, when evaluating a term or a formula, to swap two consecutive identifiers in the stack (if they are different), that is $[\![t]\!]_{\Sigma,\Pi_1\cdot(id_1,v_1)\cdot(id_2,v_2)\cdot\Pi_2} = [\![t]\!]_{\Sigma,\Pi_1\cdot(id_2,v_2)\cdot(id_1,v_1)\cdot\Pi_2}$. The first of these lemmas, for terms, is written as [6]

```
lemma eval_swap_term: forall t:term, sigma:env, pi l:stack, id1 id2:ident, v1 v2:value.
  id1 ≠ id2 →
  eval_term sigma (l++(Cons (id1,v1) (Cons (id2,v2) pi))) t =
    eval_term sigma (l++(Cons (id2,v2) (Cons (id1,v1) pi))) t
```

and is, again, proved by structural induction (See Figure 11). There is only one sub-goal which is not automatically discharged, the one corresponding to the case `Tvar`. This case is actually subtle, because it requires another induction, on the list `l`. In Coq, we use `induction l` tactic, and we finish the 2 sub-goals with `why3` tactic.

[6] Symbol ++ denotes concatenation of lists.

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Z3 (3.2)	Z3 (4.2)
lemma `eval_msubst_term`						
transformation `induction_ty_lex`						
transformation `split_goal_wp`						
subgoal 1	0.04	0.03	0.04	0.04	(5s)	(5s)
subgoal 2	0.04	0.02	0.04	0.04	(5s)	(5s)
subgoal 3	0.05	0.03	0.04	0.04	(5s)	(5s)
subgoal 4	0.12	0.03	(5s)	(5s)	(5s)	(5s)

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `eval_msubst`							
transformation `induction_ty_lex`							
transformation `split_goal_wp`							
subgoal 1	0.05	0.02	(30s)	(30s)		(30s)	(30s)
subgoal 2	0.04	0.02	(30s)	(30s)		(30s)	(30s)
subgoal 3	0.20	0.05	(30s)	(30s)		(30s)	(1000M)
subgoal 4	0.06	0.04	(30s)	(30s)		(30s)	(30s)
subgoal 5	0.04	0.03	(30s)	(30s)		(30s)	(30s)
subgoal 6	0.04	0.03	(30s)	(30s)		(30s)	(30s)
subgoal 7	0.07	0.03	(30s)	(30s)		(30s)	(1000M)
subgoal 8	0.03	0.04	(30s)	(30s)		(30s)	(30s)
subgoal 9	(5s)	4.15	(30s)	(30s)	0.78	(30s)	(1000M)
subgoal 10	0.33	0.11	(30s)	(30s)		(30s)	(30s)
subgoal 11	(5s)	(30s)	(30s)	(30s)	1.99	(30s)	(30s)
subgoal 12	2.55	0.80	(30s)	(30s)		(30s)	(1000M)

Figure 10: Substitution lemmas

```
destruct t;auto.
intros; simpl.
induction l.
(* l = Nil *)
ae.
(* Cons *)
simpl.
destruct a.
destruct (ident_decide i0 i); ae.
```

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `eval_swap_term`							
transformation `induction_ty_lex`							
transformation `split_goal_wp`							
subgoal 1	0.04	0.03	0.04	0.08		(1000M)	(1000M)
subgoal 2	(5s)	(30s)	(30s)	(30s)	1.26	(1000M)	(1000M)
subgoal 3	0.05	0.03	0.06	0.05		(1000M)	(1000M)
subgoal 4	0.18	0.07	19.46	24.35		(1000M)	(1000M)

Figure 11: Swap lemma for terms

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `eval_swap_gen`							
transformation `induction_ty_lex`							
transformation `split_goal_wp`							
subgoal 1	0.16	0.07	0.41	0.46		(30s)	(30s)
subgoal 2	0.15	0.06	0.44	0.47		(30s)	(30s)
subgoal 3	0.38	0.13	8.86	8.79		(30s)	(30s)
subgoal 4	0.38	0.14	4.20	9.18		(30s)	(30s)
subgoal 5	0.17	0.07	4.68	4.82		(30s)	(30s)
subgoal 6	0.16	0.07	5.05	5.21		(30s)	(30s)
subgoal 7	0.04	0.04	4.19	9.03		(30s)	(30s)
subgoal 8	0.04	0.04	8.51	8.96		(30s)	(30s)
subgoal 9	1.14	0.39	(30s)	(30s)		(30s)	(30s)
subgoal 10	1.07	0.41	(30s)	(30s)		(30s)	(30s)
subgoal 11	8.81	(30s)	(30s)	(30s)	1.93	(30s)	(30s)
subgoal 12	8.88	(30s)	(30s)	(30s)	1.86	(30s)	(30s)

Figure 12: Generic swap lemma for formulas

Next, we pose a similar lemma for formulas as follows

```
lemma eval_swap_gen:
  forall f:fmla, sigma:env, pi l:stack, id1 id2:ident, v1 v2:value.
    id1 ≠ id2 →
    (eval_fmla sigma (l++(Cons (id1,v1) (Cons (id2,v2) pi))) f ↔
    eval_fmla sigma (l++(Cons (id2,v2) (Cons (id1,v1) pi))) f)
```

It is again proved by structural induction (See Figure 12). Sub-goals are proved automatically. The last two cases, corresponding to the `Fforall` (2 cases because equivalence ↔ is split in two), are proved only by Alt-Ergo 0.93.1, using almost 10 seconds. We also made these proofs in Coq, that are identical, and execute more quickly:

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `eval_swap`	(5s)	(5s)	(5s)	(5s)	0.52	(5s)	(5s)

Figure 13: Specialized swap property for formulas

```
destruct f; auto.
simpl; intros.
destruct d; intros; rewrite Cons_append; ae.
```

We also pose the following instance of `eval_swap_gen` for the case `l=Nil`.

```
lemma eval_swap:
  forall f:fmla, sigma:env, pi:stack, id1 id2:ident, v1 v2:value.
    id1 ≠ id2 →
    (eval_fmla sigma (Cons (id1,v1) (Cons (id2,v2) pi)) f ↔
    eval_fmla sigma (Cons (id2,v2) (Cons (id1,v1) pi)) f)
```

While it is sufficient to instantiate the preceding lemma, this lemma is not proved by any theorem prover (see Figure 13). It has to be done in Coq

```
intros f sigma pi id1 id2 v1 v2 h1.
apply eval_swap_gen with (l:=Nil); auto.
```

Finally, two last lemmas allow to pop from the stack an identifier that does not appear in the evaluated term (resp. formula), that is $[\![t]\!]_{\Sigma,(id,v)\cdot\Pi} = [\![t]\!]_{\Sigma,\Pi}$ if id is fresh.

```
lemma eval_term_change_free : forall t:term, sigma:env, pi:stack, id:ident, v:value.
  fresh_in_term id t → eval_term sigma (Cons (id,v) pi) t = eval_term sigma pi t

lemma eval_change_free : forall f:fmla, sigma:env, pi:stack, id:ident, v:value.
  fresh_in_fmla id f → (eval_fmla sigma (Cons (id,v) pi) f ↔ eval_fmla sigma pi f)
```

Once again, the proof is done by structural induction (See Figure 14). The only sub-case which is not automatically proved is the case of `Fforall` in the direct way of implication, that is proved with:

```
destruct f; auto.
simpl; intros H sigma pi id v (h1 & h2).
destruct d; intros; rewrite <- (H _ _ id v); ae.
```

4 Hoare Logic

This section is in fact not needed for the remaining of the case study. We just illustrate how one can formalize the standard Hoare logic rules. The notion of partial correctness of a Hoare triple, usually denoted $\{p\}s\{q\}$, is defined as follows.

```
predicate valid_triple (p:fmla) (s:stmt) (q:fmla) =
    forall sigma:env, pi:stack. eval_fmla sigma pi p →
      forall sigma':env, pi':stack, n:int.
        many_steps sigma pi s sigma' pi' Sskip n →
          eval_fmla sigma' pi' q
```

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Z3 (3.2)	Z3 (4.2)
lemma `eval_term_change_free`						
transformation `induction_ty_lex`						
transformation `split_goal_wp`						
subgoal 1	0.04	0.03	0.04	0.05	(20s)	(20s)
subgoal 2	0.04	0.03	0.11	0.21	(20s)	(20s)
subgoal 3	0.04	0.03	0.03	0.04	(1000M)	(30s)
subgoal 4	0.07	0.03	3.31	3.76	(1000M)	(30s)

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `eval_change_free`							
transformation `induction_ty_lex`							
transformation `split_goal_wp`							
subgoal 1	0.05	0.04	0.20	0.25		(1000M)	(30s)
subgoal 2	0.06	0.04	0.18	0.15		(1000M)	(30s)
subgoal 3	0.08	0.04	2.01	2.05		(1000M)	(30s)
subgoal 4	0.12	0.05	2.49	2.77		(1000M)	(30s)
subgoal 5	0.05	0.03	1.07	1.09		(20s)	(20s)
subgoal 6	0.06	0.04	1.33	1.46		(30s)	(30s)
subgoal 7	0.07	0.04	2.12	2.28		(1000M)	(30s)
subgoal 8	0.04	0.03	2.71	2.62		(1000M)	(30s)
subgoal 9	0.63	0.18		(30s)	0.55	(20s)	(20s)
subgoal 10	0.23	0.08	(30s)	(30s)	0.54	(20s)	(20s)
subgoal 11	(30s)	(30s)	(30s)	(30s)	1.63	(30s)	(30s)
subgoal 12	3.02	0.37	(30s)	(30s)	1.11	(30s)	(30s)

Figure 14: Substitution and freshness

First, the consequence rule below can be proved automatically.

```
lemma consequence_rule:
  forall p p' q q':fmla, s:stmt.
  valid_fmla (Fimplies p' p) →
  valid_triple p s q →
  valid_fmla (Fimplies q q') →
  valid_triple p' s q'
```

The rule for sequence is also proved automatically, but only after posing the standard "sequence lemma":

```
lemma many_steps_seq:
    forall sigma1 sigma3:env, pi1 pi3:stack, s1 s2:stmt, n:int.
      many_steps sigma1 pi1 (Sseq s1 s2) sigma3 pi3 Sskip n →
      exists sigma2:env, pi2:stack, n1 n2:int.
        many_steps sigma1 pi1 s1 sigma2 pi2 Sskip n1 ∧
        many_steps sigma2 pi2 s2 sigma3 pi3 Sskip n2 ∧
        n = 1 + n1 + n2

lemma seq_rule:
  forall p q r:fmla, s1 s2:stmt.
  valid_triple p s1 r ∧ valid_triple r s2 q →
  valid_triple p (Sseq s1 s2) q
```

The proof of that lemma, even on paper, is not completely trivial and deserves a few lines, with an induction on the number n of steps. The proof is thus done in Coq, and is not surprising:

```
intros sigma1 sigma3 pi1 pi3 s1 s2 n Hred.
generalize Hred.
generalize (steps_non_neg _ _ _ _ _ _ _ Hred).
clear Hred.
intros H.
generalize sigma1 pi1 s1; clear sigma1 pi1 s1.
pattern n; apply Z_lt_induction; auto.
intros.
inversion Hred; subst; clear Hred.
inversion H1; subst; clear H1.
(* case s1 ≠ Sskip *)
assert (h:(0 ≤ n0 < n0+1)%Z).
  generalize (steps_non_neg _ _ _ _ _ _ _ H2); omega.
generalize (H0 n0 h _ _ _ H2).
intros (s4 & p4 & n4 & n5 & h1 & h2 & h3).
exists s4. exists p4. exists (n4+1)%Z. exists n5.
ae.

(* case s1 = Sskip *)
exists sigma2. exists pi2. exists 0%Z. exists n0.
ae.
```

The other rules are

```
lemma skip_rule:
  forall q:fmla. valid_triple q Sskip q
```

```
lemma assign_rule:
  forall p:fmla, x:mident, id:ident, t:term.
  fresh_in_fmla id p →
  valid_triple (Flet id t (msubst p x id)) (Sassign x t) p
```

```
lemma if_rule:
  forall t:term, p q:fmla, s1 s2:stmt.
  valid_triple (Fand p (Fterm t)) s1 q ∧
  valid_triple (Fand p (Fnot (Fterm t))) s2 q →
  valid_triple p (Sif t s1 s2) q
```

```
lemma assert_rule:
  forall f p:fmla. valid_fmla (Fimplies p f) →
  valid_triple p (Sassert f) p
```

```
lemma while_rule:
  forall e:term, inv:fmla, i:stmt.
  valid_triple (Fand (Fterm e) inv) i inv →
  valid_triple inv (Swhile e inv i) (Fand (Fnot (Fterm e)) inv)
```

In all these five cases, Coq must be used, and are done quite smoothly by inversion of predicates `many_steps` and `one_step`, and finishing the sub-goals with the Why3 tactic.

```
Theorem skip_rule : forall (q:fmla), (valid_triple q Sskip q).
intros q sigma pi.
intros H sigma' pi' n H1.
inversion H1; subst; clear H1; auto.
inversion H0.
Qed.
```

```
Theorem assign_rule : forall (p:fmla) (x:mident) (id:ident) (t:term),
  (fresh_in_fmla id p) → (valid_triple (Flet id t (msubst p x id))
  (Sassign x t) p).
intros p x id t h1.
red; intros.
inversion H0; subst; clear H0.
inversion H1; subst; clear H1.
inversion H2; subst; clear H2.
ae.
ae.
Qed.
```

```
Theorem if_rule : forall (t:term) (p:fmla) (q:fmla) (s1:stmt) (s2:stmt),
  ((valid_triple (Fand p (Fterm t)) s1 q) ∧ (valid_triple (Fand p
  (Fnot (Fterm t))) s2 q)) → (valid_triple p (Sif t s1 s2) q).
unfold valid_triple.
intros t p q s1 s2 (h1,h2).
intros.
inversion H0; subst; clear H0.
inversion H1; subst; clear H1.
eapply h1; eauto.
ae.
eapply h2; eauto.
ae.
Qed.
```

```
Theorem assert_rule : forall (f:fmla) (p:fmla), (valid_fmla (Fimplies p
  f)) → (valid_triple p (Sassert f) p).
unfold valid_triple.
intros f p h1.
intros.
inversion H0; subst; clear H0.
inversion H1; subst; clear H1.
inversion H2; subst; clear H2; auto.
inversion H0.
Qed.
```

```
Theorem while_rule : forall (e:term) (inv:fmla) (i:stmt),
  (valid_triple (Fand (Fterm e) inv) i inv) → (valid_triple inv (Swhile e
  inv i) (Fand (Fnot (Fterm e)) inv)).
unfold valid_triple.
intros e inv i Hinv_preserved.
intros s p Hinv_init s' p' n Hred.
generalize (steps_non_neg _ _ _ _ _ _ _ Hred); intro Hn_pos.
generalize Hred; clear Hred.
generalize s p Hinv_init; clear s p Hinv_init.
apply Z_lt_induction
 with (P := fun n =>
  forall s p,
  eval_fmla s p inv →
  many_steps s p (Swhile e inv i) s' p' Sskip n →
  eval_fmla s' p' (Fand (Fnot (Fterm e)) inv)
); auto.
intros.
inversion H1; subst; clear H1.
inversion H2; subst; clear H2.
destruct H11 as (H4 & H5).
(* case cond true *)
generalize (many_steps_seq _ _ _ _ _ _ _ H3).
intros (s3 & p3 & n1 & n2 & h1 & h2 & h3).
apply H with (3:=h2); auto.
generalize (steps_non_neg _ _ _ _ _ _ _ h1).
generalize (steps_non_neg _ _ _ _ _ _ _ h2).
now (auto with zarith).
apply Hinv_preserved with (2:=h1); simpl; auto.
(* case cond false *)
inversion H3; subst.
destruct H11 as (H4 & H5).
simpl; rewrite H5; intuition.
discriminate.
now inversion H1.
Qed.
```

The summary of all proofs done in this theory is shown on Figure 15

Proof obligations	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.0)
lemma `many_steps_seq`	1.16		
lemma `consequence_rule`		0.27	0.26
lemma `skip_rule`	0.66		
lemma `assign_rule`	1.70		
lemma `seq_rule`		0.24	0.24
lemma `if_rule`	1.11		
lemma `assert_rule`	0.86		
lemma `assert_rule_ext`	0.76		
lemma `while_rule`	0.84		

Figure 15: Proof results for theory `Hoare`

5 Weakest Precondition Calculus

5.1 Definition of the calculus

The Weakest Precondition Calculus is a function that, given a statement s and a formula Q, returns another formula denoted $\mathsf{WP}(s, Q)$. It is defined recursively on the structure of s. The usual property expected is that the Hoare triple $\{\mathsf{WP}(s, Q)\}s\{Q\}$ is valid, in other words if one executes s in a state satisfying $\mathsf{WP}(s, Q)$, and if this execution terminates, then Q is valid in the resulting state. In fact, we are going to state and prove a stronger property in Section 5.3, that also states something in case of non-termination.

```
function wp (s:stmt) (q:fmla) : fmla =
  match s with
  | Sskip               → q
  | Sassert f           → Fand f (Fimplies f q)    (** asymmetric and *)
  | Sseq s1 s2          → wp s1 (wp s2 q)
  | Sassign x t         → let id = fresh_from q in Flet id t (msubst q x id)
  | Sif t s1 s2         → (* (t → WP(s1,Q)) ∧ (not t → WP(s2,Q)) *)
      Fand (Fimplies (Fterm t) (wp s1 q)) (Fimplies (Fnot (Fterm t)) (wp s2 q))
  | Swhile cond inv body →
      (* inv ∧ forall effects, (cond ∧ inv → WP(body,inv)) ∧ (not cond ∧ inv → Q) *)
      Fand inv (abstract_effects body
                  (Fand (Fimplies (Fand (Fterm cond) inv) (wp body inv))
                    (Fimplies (Fand (Fnot (Fterm cond)) inv) q)))
  end
```

Our definition above follows the classical schema for such a calculus. For the case of `Sassert f`, we use $f \wedge (f \rightarrow Q)$ instead of $f \wedge Q$ as it is done in practice[7]. For an assignment `Sassign x t`, we substitute the mutable variable `x` by a fresh logic variable id, computed thanks to a auxiliary function `fresh_from`, then we bind id to the value assigned `t`. The case of the loop classically introduces a quantification on the side effects of the body of that loop. This quantification is realized by calling another auxiliary function `abstract_effects`.

In order to deal with difficulties separately, we do not detail implementations of those auxiliary functions, we only *axiomatize* them. The first function is axiomatized by:

```
function fresh_from (f:fmla) : ident
axiom fresh_from_fmla: forall f:fmla. fresh_in_fmla (fresh_from f) f
```

[7] It forces f to appear as hypothesis in the other proof obligations.

In other words, for any formula f, `fresh_from f` returns an arbitrary identifier that is fresh in f ; we don't care about how it is computed.

The function that abstract the side effects has type

```
function abstract_effects (s:stmt) (f:fmla) : fmla
```

and its expected behavior is, for given statement s and formula P, to return the formula $\forall w, f$ where $w = w_1, \ldots, w_k$ is the set of assigned variables in s. Our axiomatization of this behavior is made of four hypotheses detailed below.

The first hypothesis formalizes the specialization property: if $\Sigma, \Pi \models \forall w, f$ then $\Sigma, \Pi \models f$.

```
axiom abstract_effects_specialize : forall sigma:env, pi:stack, s:stmt, f:fmla.
  eval_fmla sigma pi (abstract_effects s f) → eval_fmla sigma pi f
```

The second hypothesis formalizes the distributivity of quantification over conjunction : if $\Sigma, \Pi \models (\forall w, P) \wedge (\forall w, Q)$ then $\Sigma, \Pi \models \forall w, P \wedge Q$.

```
axiom abstract_effects_distrib_conj : forall s:stmt, p q:fmla, sigma:env, pi:stack.
  eval_fmla sigma pi (abstract_effects s p) ∧
  eval_fmla sigma pi (abstract_effects s q) →
    eval_fmla sigma pi (abstract_effects s (Fand p q))
```

The third hypothesis is a bit more subtle: it expresses a property of the quantification on the two parts of an implication.

```
axiom abstract_effects_monotonic : forall s:stmt, p q:fmla.
  valid_fmla (Fimplies p q) → forall sigma:env, pi:stack.
    eval_fmla sigma pi (abstract_effects s p) → eval_fmla sigma pi (abstract_effects s q)
```

In other words, if $\models P \rightarrow Q$ then $\models (\forall w, P) \rightarrow (\forall w, Q)$. This property can be seen as a consequence of two other properties: first, if $\models f$ then $\models \forall w, f$; second, if $\forall w, (P \rightarrow Q)$ then $(\forall w, P) \rightarrow (\forall w, Q)$. It is important to notice that this hypothesis talks about validity in all states, because for fixed Σ and Π, $\Sigma, \Pi \models P \rightarrow Q$ does not imply $\Sigma, \Pi \models (\forall w, P) \rightarrow (\forall w, Q)$. A counter-example is as follows: assuming Σ such that $\Sigma(x) = 42$, then $true \rightarrow x = 42$ is valid, but not $(\forall x, true) \rightarrow (\forall x, x = 42)$.

Until now, the hypotheses on `abstract_effects` express abstractly that it behaves as an universal quantification, without saying precisely on what. The fourth and last hypothesis we pose now expresses on what variables we quantify: if $w = w_1, \ldots, w_k$ is the set of variables assigned by s, then $\forall w, P$ is a formula that is *invariant* by the WP calculus through s.

```
axiom abstract_effects_writes : forall sigma:env, pi:stack, s:stmt, q:fmla.
  eval_fmla sigma pi (abstract_effects s q) →
    eval_fmla sigma pi (wp s (abstract_effects s q))
```

In fact, this abstract way of specifying the behavior `abstract_effects` is precisely the important property that one tries to obtain when introducing the universal quantification into the weakest precondition of a loop: the formula $cond \wedge inv \rightarrow \mathsf{WP}(body, inv)$ must be quantified, so that it becomes true (because it is invariant) at each iteration of the loop.

5.2 Basic Properties of the Calculus

In the literature, there exist advanced theoretical studies on the family of *predicate transformers* in general, that includes the weakest precondition calculus [2]. Several properties are stated, often without showing why they are useful. In our case study, we are going to state and prove two properties that are needed for the main proof of soundness.

The first of these two properties is classically named *monotonicity* of the calculus. It is stated mathematically under the form: for any statement s and any formulas P and Q, if $\models P \rightarrow Q$ then $\models \mathsf{WP}(s, P) \rightarrow \mathsf{WP}(s, Q)$. In Why3 it becomes:

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `monotonicity`							
transformation `induction_ty_lex`							
transformation `split_goal_wp`							
subgoal 1	0.04	0.04	0.06	0.08		0.09	0.10
subgoal 2	(5s)	(5s)	(5s)	(5s)	1.05	(5s)	(5s)
subgoal 3	0.06	0.05	0.11	0.14		(5s)	(5s)
subgoal 4	(5s)	(5s)	(5s)	(5s)	0.97	(5s)	(5s)
subgoal 5	(5s)	(5s)	(5s)	(5s)		1.40	1.91
subgoal 6	(5s)	(5s)	(5s)	(5s)	0.75	(5s)	(5s)

Figure 16: Monotonicity of Weakest Preconditions

```
lemma monotonicity: forall s:stmt, p q:fmla.
  valid_fmla (Fimplies p q) → valid_fmla (Fimplies (wp s p) (wp s q))
```

Notice that, as before, quantification on all states is essential: $\Sigma, \Pi \models P \rightarrow Q$ does not necessarily imply $\Sigma, \Pi \models \mathsf{WP}(s, P) \rightarrow \mathsf{WP}(s, Q)$. There is a similar counter-example: if $\Sigma(x) = 42$ then $(true \rightarrow x = 42)$ but $\mathsf{WP}(x := 7, true) = true$ does not imply $\mathsf{WP}(x := 7, x = 42) = (7 = 42)$.

The proof of the monotonicity property is done by induction on the structure of s. In Why3, after application of the transformation for structural induction, it gives 6 sub-goals, as shown on Figure 16.

Only the cases of the assignment, the conditional and the loop are not proved automatically. The proofs for the case of `Sassigns` and `Sif` can both be done using the 2 same lines:

```
destruct s; auto.
unfold valid_fmla; simpl; ae.
```

it should be noted that that last call to the `why3` tactic makes use of lemmas for the substitution and fresh variables. The case of `Swhile` est slightly more involved, one needs to apply the hypothesis (3):

```
destruct s; auto.
unfold valid_fmla; simpl.
intros H1 p q H2; intuition.
apply abstract_effects_monotonic with (2:=H3).
unfold valid_fmla; simpl.
intuition.
```

The second property that we need is the *distributivity over the conjunction*: if $\Sigma, \Pi \models \mathsf{WP}(s, P)$ and $\Sigma, \Pi \models \mathsf{WP}(s, Q)$ then $\Sigma, \Pi \models \mathsf{WP}(s, P \wedge Q)$.

```
lemma distrib_conj: forall s:stmt, sigma:env, pi:stack, p q:fmla.
  eval_fmla sigma pi (wp s p) ∧ eval_fmla sigma pi (wp s q) →
  eval_fmla sigma pi (wp s (Fand p q))
```

The reciprocal is also true, but not needed. The proof proceeds again by structural induction on s, as shown on Figure 17. The cases of assignment, sequence, and loop are not proved automatically, the proofs are terminated in Coq.

The proof for the case of assignment is again easy:

```
destruct s; auto.
unfold valid_fmla; simpl; ae.
```

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `distrib_conj`							
transformation `induction_ty_lex`							
transformation `split_goal_wp`							
subgoal 1	0.06	0.04	0.06	0.07		0.14	0.12
subgoal 2	(5s)	(5s)	(5s)	(5s)	1.38	(5s)	(5s)
subgoal 3	(5s)	(5s)	(5s)	(5s)	1.05	(5s)	(5s)
subgoal 4	2.33	0.16	(5s)	(5s)		(5s)	(5s)
subgoal 5	0.53	0.08	(5s)	(5s)		(5s)	(5s)
subgoal 6	(5s)	(5s)	(5s)	(5s)	0.70	(5s)	(5s)

Figure 17: Distributivity over Conjunction

The proof in the case of the sequence is more complex. We need to use the preceding property of monotonicity. It indeed corresponds to an "intelligent" part of the proof, where we state as an intermediate lemma that $\models \mathsf{WP}(s_1, \mathsf{WP}(s_2, P) \wedge \mathsf{WP}(s_2, Q)) \rightarrow \mathsf{WP}(s_1, \mathsf{WP}(s_2, P \wedge Q))$.

```
destruct s; auto.
simpl; intros H1 H2 sigma pi p q (H3 & H4).
assert (H: valid_fmla
        (Fimplies (Fand (wp s2 p) (wp s2 q)) (wp s2 (Fand p q)))).
  unfold valid_fmla ; simpl; ae.
generalize (monotonicity s1 _ _ H).
unfold valid_fmla; simpl; ae.
```

In the case of the loop, we use hypotheses (3) and (4).

```
destruct s; auto.
simpl.
intros H sigma pi p q ((h1 & h2) & (h3 & h4)).
split; auto.
apply abstract_effects_monotonic with (p:=
(Fand (Fand (Fimplies (Fand (Fterm t) f) (wp s f))
            (Fimplies (Fand (Fnot (Fterm t)) f) p))
  (Fand (Fimplies (Fand (Fterm t) f) (wp s f))
            (Fimplies (Fand (Fnot (Fterm t)) f) q)))).
unfold valid_fmla; simpl.
intuition.
apply abstract_effects_distrib_conj; auto.
```

5.3 Soundness of the Weakest Precondition Calculus

We follow the classical approach for proving *type soundness*. It amounts to state two lemmas: on the one hand, the preservation of WP by reduction ; on the other hand a *progress* property, that is the validity of $\mathsf{WP}(s, Q)$ entails the reductibility of s (if $s \neq$ `Sskip`).

```
lemma wp_preserved_by_reduction: forall sigma sigma':env, pi pi':stack, s s':stmt.
  one_step sigma pi s sigma' pi' s' →
  forall q:fmla. eval_fmla sigma pi (wp s q) → eval_fmla sigma' pi' (wp s' q)
```

Proof obligations	Coq (8.3pl4)
lemma `wp_preserved_by_reduction`	6.78

Figure 18: Preservation by reduction

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `progress`							
transformation `induction_ty_lex`							
transformation `split_goal_wp`							
subgoal 1	0.04	0.03	0.05	0.06		0.00	0.00
subgoal 2	(5s)	(5s)	(5s)	(5s)	0.56	(5s)	(5s)
subgoal 3	(5s)	(5s)	(5s)	(5s)	1.12	(5s)	(5s)
subgoal 4	(5s)	(5s)	(5s)	(5s)	0.73	(5s)	(5s)
subgoal 5	(5s)	(5s)	(5s)	(5s)	0.70	(5s)	(5s)
subgoal 6	(5s)	(5s)	(5s)	(5s)	0.75	(5s)	(5s)

Figure 19: Progress lemma

This first lemma is proved by induction on hypothesis `(one_step sigma pi s sigma' pi' s')`. This must be done in Coq (Figure 18). After using the `induction` tactic in Coq, there are 8 sub-goals, among which 6 can be proved simply by the `simpl` tactic followed by the `why3` tactic. The two remaining cases are for the reduction of loops. To prove them, we start by using the hypothesis (1) to make explicit the fact that the formula $(cond \wedge inv \rightarrow \mathsf{WP}(body, inv)) \wedge (\neg cond \wedge inv \rightarrow Q)$ is true in the current state, then we conclude using the `why3` tactic. One should notice that these calls to `why3` tactic use the hypothesis (4) and the property `distrib_conj`.

```
intros sigma sigma' pi pi' s s' h1.
induction h1; try (simpl; intro; ae).
(* case while true do ... *)
simpl; intros q (_ & h).
(* need to keep a copy of h *)
generalize h; intro h'.
apply abstract_effects_specialize in h'; simpl in h'; ae.
(* case while false do ... *)
simpl; intros q (_ & h).
apply abstract_effects_specialize in h; simpl in h; ae.
```

The progress lemma is stated as follows.

```
lemma progress: forall s:stmt, sigma:env, pi:stack,
  sigmat: type_env, pit: type_stack, q:fmla.
    compatible_env sigma sigmat pi pit ∧ type_stmt sigmat pit s ∧
    eval_fmla sigma pi (wp s q) ∧ s ≠ Sskip → reducible sigma pi s
```

It may seem strange that the lemma above depends on an arbitrary formula Q, that does not play any role in the conclusion. Thus, it may seem enough to state this lemma for $Q = true$. But then it would not be general enough to be proved by induction. For example, to prove the progress of a sequence $s_1; s_2$ knowing that $\mathsf{WP}(s_1; s_2, true)$ holds,

we need to prove the progress of s_1 knowing that $\mathsf{WP}(s_1, \mathsf{WP}(s_2, true))$ holds, and the formula $\mathsf{WP}(s_2, true)$ is *a priori* arbitrary.

The progress lemma is proved by structural induction on s. Only the case of `Sskip` is then proved automatically (Figure 19), the reason being that the conclusion `reducible` is existentially quantified. In each case we terminate the proof in Coq, in a few lines, with the `exists` tactic. For the case of the loop, we use once more the property `distrib_conj`.

```
destruct s; auto.
intros.
do 3 eexists; econstructor; eauto.
```

```
destruct s; auto.
intros.
destruct (decide_is_skip s1).
(* case s1 = Sskip *)
subst s1.
do 3 eexists.
apply one_step_seq_skip.
(* case s1 ≠ Sskip *)
inversion H2; subst; auto.
destruct H0 with (1:=H1) (q:= (wp s2 q)) as (sigma2 & pi2 & s3 & h3); auto.
exists sigma2. exists pi2. exists (Sseq s3 s2). ae.
```

```
destruct s; auto.
intros.
inversion H2; subst; clear H2.
unfold reductible.
apply eval_type_term with (sigma := sigma) (pi := pi) in H10 ; auto.
assert (eval_term sigma pi t = Vbool true  ∨
        eval_term sigma pi t = Vbool false).
generalize (type_inversion (eval_term sigma pi t)).
destruct (eval_term sigma pi t); simpl;  try discriminate.
destruct b; auto.
destruct H2.
do 3 eexists.
apply one_step_if_true; auto.
do 3 eexists.
apply one_step_if_false; auto.
```

```
destruct s; auto.
simpl.
intros sigma pi sigmat pit q H H1 (H2 & H3) H4.
do 3 eexists.
apply one_step_assert; auto.
```

Proof obligations	Alt-Ergo (0.93.1)	Alt-Ergo (0.94)	CVC3 (2.2)	CVC3 (2.4.1)	Coq (8.3pl4)	Z3 (3.2)	Z3 (4.2)
lemma `wp_soundness`	(5s)	(5s)	(5s)	(5s)	0.70	(5s)	(5s)

Figure 20: Proof of main result

```
destruct s; auto.
simpl.
intros H sigma pi sigmat pit q H1 H2 (H3 & H4) H5.
inversion H2; subst; clear H2.
unfold reductible.
apply eval_type_term with (sigma := sigma) (pi := pi) in H11 ; auto.
assert (eval_term sigma pi t = Vbool true  ∨
        eval_term sigma pi t = Vbool false).
generalize (type_inversion (eval_term sigma pi t)).
destruct (eval_term sigma pi t); simpl;  try discriminate.
destruct b; auto.
destruct H0.
do 3 eexists.
apply one_step_while_true; auto.
do 3 eexists.
apply one_step_while_false; auto.
```

Our main result is now: for any program s, any state Σ, Π and any formula Q, if $\Sigma, \Pi \models \mathsf{WP}(s, Q)$ then either s executes infinitely, or it reduces to `Sskip`, and in the second case Q is true in the final state. Since it is not handy to formulate that an execution is infinite, we state that slightly differently: if s executes into some finite number of steps into s' such that s' does not reduce anymore, then $s' = \texttt{Sskip}$. All this must be stated only for well-typed programs, thus the Why3 formulation of our main result is:

```
lemma wp_soundness: forall n :int, sigma sigma':env, pi pi':stack, s s':stmt,
  sigmat: type_env, pit: type_stack, q:fmla.
    compatible_env sigma sigmat pi pit ∧ type_stmt sigmat pit s ∧
    many_steps sigma pi s sigma' pi' s' n ∧ not (reducible sigma' pi' s') ∧
    eval_fmla sigma pi (wp s q) → s' = Sskip ∧ eval_fmla sigma' pi' q
```

This is proved by induction on n, what we do calling Coq (Figure 20). For $n = 0$, the progress lemma ensures that s' can only be `Sskip`. For the case $n > 0$, we know that s reduces in one step into some s_0, the lemma of preservation of WP by reduction allows us to apply the induction hypothesis on s_0.

```
intros n sigma sigma' pi pi' s s' sigmat pit q.
intros Hcomp Htype (h1,(h2,h3)).
generalize (steps_non_neg _ _ _ _ _ _ _ h1).
intros Hnpos.
generalize sigma sigma' pi pi' s s' sigmat pit q Hcomp Htype h1 h2 h3.
clear sigma sigma' pi pi' s s' sigmat pit q Hcomp Htype h1 h2 h3.
generalize Hnpos.
pattern n; apply Z_lt_induction; auto.
intros x Hind Hxpos.
intros.
inversion h1; subst; clear h1.
(* cas zero etapes *)
destruct (decide_is_skip s').
subst s'.
split; auto.
contradiction h2; clear h2.
apply progress with (q:=q) (1:=Hcomp); auto.
(* cas au moins une etape *)
generalize (steps_non_neg _ _ _ _ _ _ _ H0).
intro.
generalize (type_preservation s s2 sigma sigma2 pi pi2 sigmat pit).
intros h.
apply Hind with (y:=n0) (sigmat:=sigmat) (pit:=pit) (5:=H0);
  intuition.
apply wp_preserved_by_reduction with (1:=H); auto.
```

6 General remarks on proofs

Let's put aside the section on Hoare logic which was presented for illustration but not needed for the main results. Overall, the proofs that we needed to do in Coq amount to 142 lines of tactics. This number should be compared with the number of lines of specification that we wrote: 390, that is more than twice the number of lines of proofs. This is a really good level of automation, that invalidates the usual belief that a proof of complex behavior of a program requires significantly more lines of proofs than lines of code.

The source of this case study, together with the Why3 proof session and the Coq proofs are available on the web page http://toccata.lri.fr/gallery/WP_revisited.en.html. A ZIP archive is provided, that contains everything needed to replay the proofs using Why3 0.80 [6].

7 Conclusions

We have presented a technique for proving the soundness of a weakest preconditions calculus, that imitates the classical technique of proof of type soundness of a programming language, by showing both that the WP is preserved by reduction and that the validity of the WP entails the reducibility. Since the conformance of the annotations of a program is by definition the absence of blocking (blocking semantics), we obtain a verification method, guaranteed correct, even on non-terminating programs. The definition via blocking semantics is proposed by Herms *et al.* [10] in the case of a great-step semantics, soundness being proved by co-induction. We use a small-step blocking semantics : our approach is thus original as far as we know. Indeed, a similar approach was used by Conchon and Filliâtre [8] in a different context: proof of soundness of a decision procedure for the safety of use of semi-persistent data structures.

The proof obtained by our approach is not difficult, in fact we pretend this is a natural approach, simple to understand and moreover is done with highly automated proof. We plan to use this method for teaching the basics

of deductive program verification[8].

Along this case study, we identified several interesting features of the Why3 environment: structuring in theories, algebraic data types, inductive predicate, direct calls to automated provers, transformation for proving by structural induction, call to Coq to terminate difficult proofs, use of the `why3` tactic to terminate the Coq proofs. At the end, the number of lines of manual proof is small: 142 lines of Coq proofs for 390 lines of program code, that is a very satisfactory degree of automation. Nevertheless, one should be aware that during the development of this study, we had to do, temporarily, more Coq proofs, in order to identify the appropriate lemmas needed to finally get good automation. The `why3` tactic is by the way very useful during such a phase of development of proofs.

We also identified a few weaknesses of Why3 that could make interesting motivations for extensions. The automated provers appear to be weak as soon as a hypothesis involves an inductive predicate: a transformation to make an *inversion* of such a hypothesis, so as to reason on it by induction, could be very handy and useful. For example, the lemma `steps_non_neg` could be proved without the need of Coq. Similarly, a transformation for proving a lemma by induction on integers would be useful.

We noticed a general weakness of SMT provers: often they can't prove a goal that can be proved in a few lines of Coq. For example, these provers are not able to do a simple thing like `eexists; eauto` of Coq. We also tested some provers specialized for first-order reasoning (of the kind « TPTP » : Vampire [12], Spass [14], Eprover [13]) without success. We also noticed that the automated provers do not handle well the goals that can be proved very easily with the `simpl` tactic of Coq followed by a call to the `why3` tactic. This emphasizes a weakness in the support for *computations within proofs* by such provers.

Another kind of missing feature is the ability to pose statements about the functions of the Why3 *programming language*. In fact, in this case study we only defined pure functions. If a function like `wp` was written using the programming language (that allow side-effects), then it would not be possible to state properties of its behavior like we did, but only by giving a post-condition, and in particular it would not be possible to state a property like `wp_soundness`.

The case study itself deserves to be extended. First, the functions `fresh_from` and `abstract_effects` should be realized. A significant extension would to handle a language with several sub-programs, like it was done by Herms *et al.* [10] and was indeed a reason to use the blocking semantics approach.

In a longer term, one could seriously think about using this approach to develop code correct by construction. In fact, an extension of Why3 currently in progress is the ability to extract OCaml code. With respect to an approach fully in Coq, we would gain a lot of automation.

Acknowledgments The authors thank Jean-Christophe Filliâtre and Paolo Herms for the concept of blocking semantics to define the validity of annotated code, Levs Gondelmans who implemented the transformation of structural induction of Why3, Jean-Christophe Filliâtre and Andrei Paskevich who implemented the `why3` tactic, and also the other authors of Why3, Guillaume Melquiond and François Bobot, without whom this work would not have been achieved.

References

[1] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, number 3603 in Lecture Notes in Computer Science, pages 50–65. Springer, 2005.

[2] R.-J. Back and J. von Wright. *Refinement calculus - a systematic introduction*. Undergraduate texts in computer science. Springer, 1999.

[3] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, Berlin, Germany, July 2007. Springer.

[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[8] See `http://www.lri.fr/~marche/MPRI-2-36-1/`

[5] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/.

[6] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 platform, version 0.80*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.80 edition, Oct. 2012. https://gforge.inria.fr/docman/view.php/2990/8186/manual-0.80.pdf.

[7] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[8] S. Conchon and J.-C. Filliâtre. Semi-Persistent Data Structures. In *17th European Symposium on Programming (ESOP'08)*, Budapest, Hungary, Apr. 2008.

[9] L. de Moura and N. Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

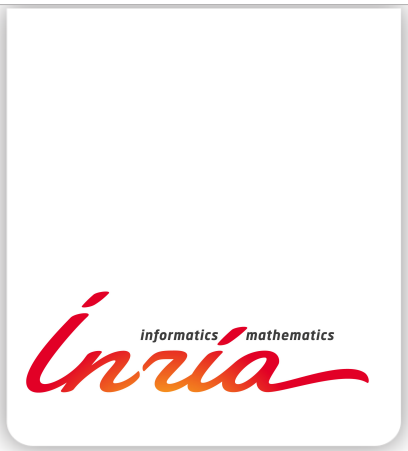
[10] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software: Theories, Tools, Experiments (4th International Conference VSTTE)*, volume 7152 of *Lecture Notes in Computer Science*, pages 2–17, Philadelphia, USA, Jan. 2012. Springer.

[11] The PVS system. http://pvs.csl.sri.com/.

[12] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 292–296, Trento, Italy, July 1999. Springer.

[13] S. Schulz. System description: E 0.81. In D. A. Basin and M. Rusinowitch, editors, *Second International Joint Conference on Automated Reasoning*, volume 3097 of *Lecture Notes in Computer Science*, pages 223–228. Springer, 2004.

[14] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In R. A. Schmidt, editor, *22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399